

Symbolic Execution For Verification

JOXAN JAFFAR, JORGE A. NAVAS, AND ANDREW E. SANTOSA

National University of Singapore
 {joxan,navas,andrews}@comp.nus.edu.sg

Abstract. In previous work, we presented a symbolic execution method which starts with a concrete model of the program but progressively abstracts away details only when these are known to be irrelevant using interpolation. In this paper, we extend the technique to handle unbounded loops. The central idea is to progressively discover the strongest invariants through a process of loop unrolling. The key feature of this technique, called the minimax algorithm, is *intelligent backtracking* which directs the search for the next invariant. We then present an analysis of the main differences between our symbolic execution method and mainstream techniques mainly based on abstract refinement (CEGAR). Finally, we evaluate our technique against available state-of-the-art systems.

1 Introduction

CounterExample-Guided Abstraction Refinement (CEGAR, or more briefly, AR) [8,2,21], has been a very successful technique for proving safety in large programs. Starting with a coarse abstraction of the program (*abstraction phase*), the abstraction is checked for the desired property (*verification phase*). If no error is found, then the program is safe. Otherwise, an abstract counterexample is produced. The counterexample is then analyzed to test if it corresponds to a concrete counterexample in the original program. If yes, the program is reported as unsafe. Otherwise, a *counterexample-driven refinement* is performed to refine the abstract model such that the abstract counterexample is excluded (*refinement phase*), and the process starts again. Several systems have been developed during recent years following this approach [1,7,14,20,9,12,3,11].

In a previous work [17] we presented a *dual* algorithm to AR, here called *Abstraction Learning*, for loop-free program fragments. Essentially, our technique starts with the concrete model of the program. Then, the model is checked for the desired property (*verification phase*) via *symbolic execution*. If a counterexample is found, then it must be a real error and hence, the program is unsafe. Otherwise, the program is safe. In order to make the symbolic execution process practical, the technique learns the facts that are irrelevant for keeping infeasible paths by computing *interpolants* (*learning phase*), and then it eliminates those facts from the model (*abstraction phase*). Unfortunately, this work did not provide an automatic treatment of loops while it assumed user-provided loop invariants to make symbolic executions finite.

In this paper, we extend the technique proposed in [17] to discover loop invariants. The central idea is to progressively discover the strongest invariants through a lazy process of loop unrolling.

For a given loop, *path-based loop invariants* are computed and used to generalize the states at the looping points (program points where the merging of control paths

construct some cyclical paths). Our computation of invariants is *lightweight* as they are computed by manipulation, using the theorem prover, of explicit constraints. The algorithm attempts to minimize the loss of information by computing the *strongest* possible invariants. These *speculative* invariants may be still too coarse to ensure safety. Here the algorithm computes interpolants to ensure that error locations are not reachable, resulting in *selective* unrolling at points where the path-based invariant can no longer be produced due to the strengthening introduced by the interpolants. Similar to AR, this procedure is only guaranteed to terminate when loop iterations are bounded.

A fundamental distinction with AR is that we attempt to always construct the most precise abstraction for loops by computing the strongest lightweight loop invariants. This feature is vital to detect as many infeasible paths as possible during the symbolic execution-based traversal. Our thesis is that this investment often pays off, and even in examples where it does not, it is affordable.

The contributions of this paper can be summarized as follows:

1. We extend the interpolation-based symbolic execution algorithm in [17] to deal with unbounded loops by describing a novel lazy loop unrolling algorithm called *minimax*.
2. We provide an analysis using several academic examples of the major differences between our proposed algorithm and mainstream techniques mainly based on abstraction refinement.
3. Finally, we implement the main ideas of this paper in a system called TRACER, and we evaluate it using real programs against BLAST, available state-of-the-art system.

Related Work. Our work is clearly related to abstraction refinement (CEGAR) [8,2,21,14,13]. We dedicate Sec. 3 to exemplify main differences through some academic examples and Sec. 6 to compare with BLAST using real programs.

Recent algorithms such as Synergy/DASH/SMASH [12,3,11] use test-generation features to enhance the process of verification. The main advantage comes from the use of lightweight symbolic execution provided by DART [10] to mitigate the expensive cost of the *abstract post-image operator* when predicate abstraction is used. An advantage of our approach is that it does not suffer from this drawback since ours is symbolic execution-based and does not use predicate abstraction. More importantly, these tools rely on CEGAR to build the abstract model of the program, and hence, major limitations observed in Sec. 3 still hold. Moreover, there is no benefit of using test cases using our method unless there is a real counterexample in the program. On the contrary, we can construct reasonable scenarios where Synergy and its descendants can have an exponential slowdown wrt to ours as shown in Sec. 3.

Our closest related works are in [16,17], where interpolation was performed on a search tree of a CLP goal in pursuit of a target property. (The earlier paper [16] focussed on a finite domain for an optimization problem.) But these works did not consider loops. The main conceptual advance of this paper is to address loops, and in doing so, allows for the consideration of real-life programs. Furthermore, this paper provides a detailed analysis of differences with the state-of-the-art CEGAR method, and finally, we present a comprehensive experimental evaluation with BLAST, the most advanced CEGAR implementation available to us at this time.

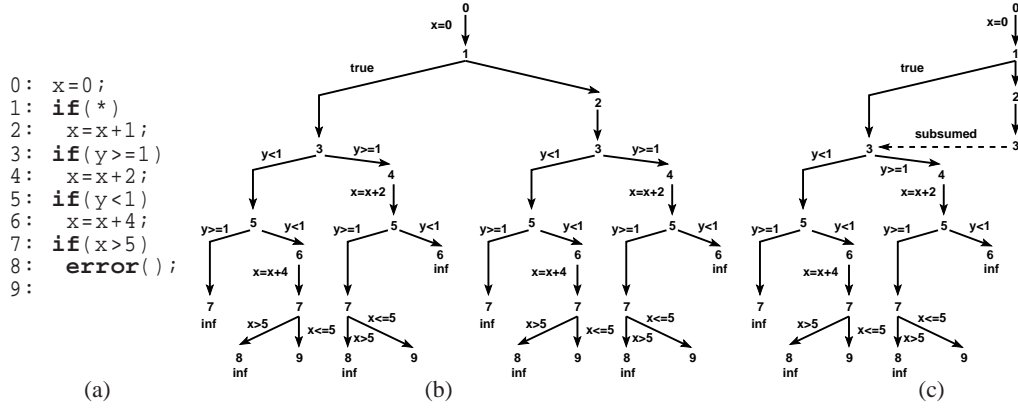


Fig. 1. Interpolation and Subsumption of Infeasible Paths

Very recently, another interpolation-based symbolic execution method has been proposed, independent from ours, in [19]. This work can be considered in two parts. In the consideration of loop-free program fragments, this work is in fact subsumed by the earlier works [16,17]. In the consideration of loops, [19] presented a *naive* strategy for handling loops based on an iterative deepening process. The central idea is to compute interpolants for a fixed depth in the hope they will converge to inductive assertions after an expensive fixpoint computation. We quote from [19]: “*the question of how to obtain convergence in practice for unbounded loops needs further study*”. Therefore, the description of a concrete algorithm from this idealistic one is far from being trivial. Furthermore, experimental evaluation was provided only in regard to testing, and not for the case of verification. In contrast, in this paper we present a *directed* approach which essentially amounts to an intelligent backtracking strategy which takes into account the reason for failure at the current stage.

2 The Basic Idea

Our basic algorithm performs *symbolic execution* of the programs while attempting to find an execution path that reaches the **error()** function. If such path cannot be found, then it concludes that the program is safe.

Consider the program in Fig. 1(a). We depict in Fig. 1(b) the naive symbolic execution tree, and in Fig. 1(c) a smaller tree, which still proves the absence of bugs. During the traversal of the tree, our algorithm *preserves the infeasibility* of the paths using the well-known concept of *interpolation*. Let us focus on Fig. 1(c) and consider, for instance, the path $A \equiv \langle 0 \rangle - \langle 1 \rangle - \langle 3 \rangle - \langle 5 \rangle - \langle 7 \rangle$ which is detected as infeasible ($x = 0 \wedge y < 1 \wedge y \geq 1$). Applying our infeasibility preservation principle, we keep node $\langle 7 \rangle$ labeled with *false*. This produces the interpolant $y < 1$ at node $\langle 5 \rangle$ since this is the most general condition that preserves the infeasibility of node $\langle 7 \rangle$. Note that here, $y < 1$ is entailed by the original state $x = 0 \wedge y < 1$ of node $\langle 5 \rangle$ and in turn entails $y \geq 1 \models \text{false}$.

Now consider another path $B \equiv \langle 0 \rangle - \langle 1 \rangle - \langle 3 \rangle - \langle 5 \rangle - \langle 6 \rangle - \langle 7 \rangle - \langle 8 \rangle$ and the node $\langle 8 \rangle$ with the formula $y < 1 \wedge x = 4 \wedge x > 5$ which is also infeasible. The node $\langle 7 \rangle$ can be interpolated to $x \leq 5$. As before, this would produce the precondition $x \leq 1$ at $\langle 5 \rangle$. The

final interpolant for $\langle 5 \rangle$ is the *conjunction* of $y < 1$ (produced from A) and $x \leq 1$ (produced from B). In this way, when $\langle 5 \rangle$ is visited through the path $\langle 0 \rangle - \langle 1 \rangle - \langle 3 \rangle - \langle 4 \rangle - \langle 5 \rangle$ the state cannot yet be subsumed since the current context $y \geq 1 \wedge x = 2$ does not entail the interpolant stored at $\langle 5 \rangle$ ($y < 1 \wedge x \leq 1$). After that, the symbolic execution continues normally until the prefix $C \equiv \langle 0 \rangle - \langle 1 \rangle - \langle 2 \rangle - \langle 3 \rangle$ is traversed. The formula $x = 0$ associated to the state at $\langle 3 \rangle$ entails the interpolant at $x \leq 1$ at $\langle 3 \rangle$ and hence, our algorithm finishes proving safety without traversing the whole subtree rooted at prefix C .

Loops. We now explain how our algorithm handles loops using a slightly modified classic example from [14] shown in Fig. 2(a). Essentially, it automatically infers path-based loop invariants using information learned during traversal. The constructed loop invariant for a given path inside a loop is a conjunction of constraints whose truth values remain unchanged after one or more iterations of the loop. Similar to abstraction refinement, this process may require refinements in the case the abstraction is too coarse to prove the safety property.

In Fig. 2(b) assume the first path explored is $\langle 0 \rangle - \langle 1 \rangle - \langle 2 \rangle - \langle 3 \rangle - \langle 1' \rangle$ denoting a cyclic path from location $\langle 1' \rangle$ back to $\langle 1 \rangle$. Note that $\langle 1' \rangle$ and $\langle 1 \rangle$ correspond to the same program point. We use primed versions to distinguish multiple occurrences. Our algorithm then examines the constraints at the entry of the loop (i.e., $lock == 0, new == old + 1, flag == 1$) to discover those whose truth values remain unchanged after the loop (i.e., $lock == 1, new == old, flag == 1$). Clearly, the constraints $lock == 0$ and $new == old + 1$ are no longer satisfied while $flag == 1$ still holds.

At this point, our algorithm produces an abstraction at the location $\langle 1 \rangle$ by making the truth values of $lock == 0$ and $new == old + 1$ unknown. In this way, the constraints at $\langle 1' \rangle$ now entails the modified constraints of $\langle 1 \rangle$ ($flag == 1$), achieving parent-child subsumption. Assume the next explored path is $\langle 0 \rangle - \langle 1 \rangle - \langle 2 \rangle - \langle 3 \rangle - \langle 4 \rangle - \langle 1'' \rangle$ (Fig. 2(b)). At $\langle 1'' \rangle$, the constraints already entail the generalized constraint of $\langle 1 \rangle$ (they are invariant), and we therefore stop the traversal.

After the loop is traversed, the remaining constraint at $\langle 1 \rangle$ is $flag == 1$ and this is in fact a loop invariant discovered by the algorithm. Since we have removed $new == old + 1$ from $\langle 1 \rangle$, the exit path of the loop now becomes feasible as the condition $new == old$ becomes satisfiable. For this reason the traversal reaches $\langle 5 \rangle$ with the constraint $flag == 1$ propagated from $\langle 1 \rangle$ and $new == old$ which is obtained by strongest postcondition propagation through the loop exit transition.

Since we keep $flag == 1$ in the loop invariant at $\langle 1 \rangle$, the algorithm manages to reason that the path $\langle 0 \rangle - \langle 1 \rangle - \langle 5 \rangle - \langle 6 \rangle$ is infeasible (Fig. 2(b)). One important point here is that the algorithm exits the loop with maximal information. This is useful to detect as many infeasible paths as possible. An AR algorithm would not detect the infeasibility and would visit **error()** at $\langle 8 \rangle$.

Next, our algorithm visits the nodes $\langle 7 \rangle$ and $\langle 8 \rangle$ also in Fig. 2(b), which is an error location. The path is spurious, and the algorithm discovers using interpolation that one of the reason for the reachability of this point is the removal of $new == old + 1$ at $\langle 1 \rangle$. The algorithm decides to *lock* $new == old + 1$ at $\langle 1 \rangle$ and restarts the traversal from $\langle 1 \rangle$. Locking declares that the constraint cannot be removed for generating loop invariant. This is our main mechanism to ensure *progress*.

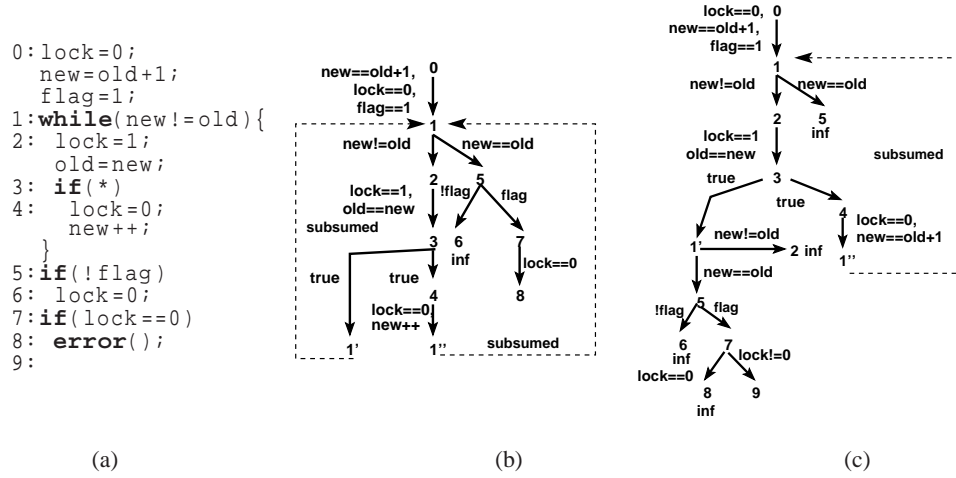


Fig. 2. Loops

The next traversal after the locking is depicted in Fig. 2(c). Similar to the first traversal, the path $\langle 1 \rangle - \langle 2 \rangle - \langle 3 \rangle - \langle 1' \rangle$ is again re-traversed. At $\langle 1' \rangle$, the constraints do not entail the constraints of $\langle 1 \rangle$ anymore. Due to locking of $\text{new} = \text{old} + 1$, we are prevented from generating a loop invariant, and hence, subsumption does not hold. As the result, the traversal continues, and it is completed without visiting the error program point at $\langle 8 \rangle$.

An essential observation is that due to its directed search for loop invariants, the algorithm does not unroll the location at $\langle 1'' \rangle$ (Fig. 2(c)) since the state is already subsumed by $\langle 1 \rangle$ without the need to force any abstraction. A naive iterative deepening algorithm (e.g., [19]) would also unroll that path, and hence, we can construct reasonable scenarios in which this leads to an exponential explosion.

3 Comparison with the State-Of-The-Art

We now analyze essential differences between our approach and mainstream techniques which are mainly based on abstraction refinement (CEGAR).

Exploration of Infeasible Paths. The core idea of abstraction refinement is to use the most general abstraction first, and refine later. This causes the exploration of *infeasible paths* which *stresses significantly* well-known problems in AR. First, the more predicates are considered in the abstract model the more costly will be the verification phase. Moreover, if predicate abstraction is used (e.g., SLAM and BLAST) expensive abstract post-image and quantifier elimination are needed. Finally, the cost of the refinement process may be also prohibitive.

Because of the huge impact of exploring infeasible paths significant research has been done recently. A partial solution has been the use of DART in order to provide a symbolic execution engine in Synergy-like tools [12,3,11]. However, the construction of the abstract model is still needed and the above problems persist. Furthermore, these tools may perform unnecessary refinements that may create reasonable scenarios which lead to an exponential behavior. Consider the program in Fig. 3(a). Assume that a Synergy-like tool produces the test case $\langle 1 \rangle - \langle 2 \rangle - \langle 3 \rangle - \langle 7 \rangle - \langle 10 \rangle$, and that the abstract model, with no predicates, reaches the error through the path $\langle 1 \rangle - \langle 2 \rangle - \langle 3 \rangle - \langle 5 \rangle - \langle 6 \rangle - \langle 8 \rangle - \langle 9 \rangle$. Then, it tries now to produce new test cases by negating the first constraint which

<pre> 1: q=1; 2: if(*) 3: x=0; else 4: x=1; 5: if(x>0) 6: y=0; else 7: y=1; 8: if(q==0) 9: error(); 10: </pre>	<pre> 1: assume(y == 0); 2: n = 0; 3: while (n < N){ 4: y++; 5: n++; 6: if (y+n < N) 7: error(); </pre>	<pre> 1: s=0; 2: if(*) z=0; 3: else z=999; // 1 4: if(*) s++; 5: else s+=2; // N 6: if(*) s++; 7: else s+=2; 8: if(s+z>2*N && z==0) 9: error(); </pre>	<pre> 1: if(*){ 2: x=0; 3: y=0; 4: else{ 5: x=complex_func(); 6: y=0; 7: s=x; 8: t=y; 9: if(*){s++;t++;} 10: // 1*/ 11: if(*){s++;t++;} 12: if(t>N && s>N) error(); </pre>
(a)	(b)	(c)	(d)

Fig. 3. Several Programs

is not in the common prefix (i.e., $x > 0$) but it is unsatisfiable since $x = 0$. Therefore, it will likely add the predicate $x \leq 0$ which is irrelevant for proving safety. Our technique will traverse the path through $\langle 1 \rangle - \langle 2 \rangle - \langle 3 \rangle - \langle 7 \rangle - \langle 8 \rangle$ and produce the interpolant $q = 1$. The rest of paths will entail that interpolant, and hence, the behavior will be linear on the size of the program.

Discovering Loop Invariants. Any symbolic traversal method will have to eventually discover loop invariants that are strong enough for the proof process to conclude successfully. In the case of AR, the abstract model is refined from spurious counterexamples by discovering which predicates can refute the error path, and in this process, they are *hoped* to be in fact invariant through loops.

A crucial observation is that the inference of invariant predicates can speedup significantly the convergence of loops [4]. We therefore employ invariant discovery by searching for the *strongest* invariants. This principle is also in accordance with our philosophy to perform concrete symbolic execution in order to maintain exact information for loop-free fragments.

Fig. 3(b) illustrates the benefits of computing strongest loop invariants. AR will discover the predicates $(n = 0), (n = 1), \dots, (n = N - 1)$ and also $(y = 0), \dots, (y = N)$, and hence full unrolling of the loop is needed. To understand why our approach avoids the full unrolling, the concept of inference of path-based loop invariant constraints is essential. Consider the path $\langle 1 \rangle - \langle 2 \rangle - \langle 3 \rangle - \langle 4 \rangle - \langle 5 \rangle - \langle 3' \rangle$. The state at $\langle 3' \rangle$ can be specified by the constraint $y = 0 \wedge n = 0 \wedge n < N \wedge y' = y + 1 \wedge n' = n + 1$ on the variables x' and y' . Our algorithm will attempt to infer which constraints are individually invariant in order to get parent-child subsumption (i.e., “close” the loop). It is straightforward to see that $y \geq 0$ (by slackening $y = 0$ to $y \geq 0$) is invariant through the loop because when $\langle 3 \rangle$ is first visited, $y = 0 \Rightarrow y \geq 0$ holds and after one iteration the constraints $y \geq 0 \wedge y' = y + 1$ still imply $y' \geq 0$. The second essential step is when the exit condition is taken (i.e., $\langle 1 \rangle - \langle 3 \rangle - \langle 6 \rangle$) our technique will attach $n \geq N$ to all invariant constraints (in this case $y \geq 0$) by computing strongest postcondition. More importantly, those two constraints ($y \geq 0 \wedge n \geq N$) suffice to prove that the error condition $y + n < N$ is false. Therefore, we are done with only one iteration through the loop.

Using Newly Discovered Predicates in Future Traversal. Another fundamental question in AR: after the set of predicates required to exclude the spurious counterexample

has been discovered, how should those predicates be used in other paths? Consider our next program in Fig 3(c).

A counterexample-guided tool will discover the predicates $(s = 0), (s = 1), \dots, (s = 2 * N)$. Then, it will either add $(z = 0)$ or $(z = 999)$. Assume that it first adds the predicate $(z = 0)$. The key observation is that all the paths that include $(z = 999)$ (location $\langle 3 \rangle$) will be traversed considering all the predicates discovered from paths that included $(z = 0)$ (location $\langle 2 \rangle$), and hence, the traversal will be exponential.

Our algorithm will basically perform the same amount of work for the case that $z = 0$ is considered. However, it traverses the paths that include $z = 999$ without consideration of the facts learnt from paths that include $z = 0$ since it only keeps track of the concrete state collected so far (i.e., $s = 0 \wedge z = 999$). Then, after the path $\langle 1 \rangle - \langle 3 \rangle - \langle 4 \rangle - \dots - \langle 6 \rangle - \langle 8 \rangle$ is traversed we can discover in a straightforward manner that $z = 999$ suffices to refute the error state and hence, the rest of the paths will be subsumed. Notice that AR will also discover the predicate $(z = 999)$ after the counterexample is found. The essential difference, for this class of programs, is that the predicates discovered previously $((s = 0), (s = 1), \dots, (s = 2 * N))$ are used, and hence, the traversal will be significantly affected by them.

Running an Abstract State Hampers Subsumption. The next example illustrates another potential weakness of AR that is not present in our approach. Even if locality is well exploited, the likelihood of subsuming the *currently traversed* state may be diminished because the state, being abstract, is too coarse. Consider now the program in Fig. 3(d). Assume `complex_func` returns always 0.

In principle, a counterexample-guided tool will behave very similarly as in the program in Fig. 3(c). Assume that the prefix path $\langle 1 \rangle - \langle 2 \rangle - \langle 3 \rangle$ is taken. It will then discover the predicates $(x = 0), (s = 0), (s = 1), \dots, (s = N), (y = 0), (t = 0), (t = 1), \dots, (t = N)$. Again, those predicates are likely to be used during the exploration of the else-branch $(\langle 4 \rangle - \langle 5 \rangle - \langle 6 \rangle)$. However, an essential difference with respect to program in Fig. 3(c) is that although the discovered predicate $(x = 0)$ is taken into consideration, the abstract state cannot be covered since it is too coarse assuming it does not consider lazily the value returned by `complex_func`, and hence, it does not entail the predicate $(x = 0)$. In contrast, since our method does perform a systematic propagation of the program state the value returned by `complex_func` will be captured and we will be able to entail the interpolant $x = 0$. The main consequence is that the state now will be subsumed.

Unnecessary Detection of Infeasible Paths. So far we have illustrated scenarios where our approach behaves better than AR. The advantage exploited in the preceding examples is the preservation of infeasible paths while abstracting loops using the strongest lightweight loop invariants. Unfortunately, this characteristic might be an important downside if the program can be proved safe even traversing infeasible paths since all the work of generating interpolants for preserving infeasible paths would be wasteful.

We claim that eager detection of infeasible paths even if they are not relevant to the safety property is not limiting in practice. The reason is that many of the infeasible paths in real programs must be considered anyway to block the error paths, and hence, counterexample-guided approaches will also consider them although lazily paying a higher price later on. The results obtained by our prototype with real programs shown in Sec. 6 support strongly our view.

To elaborate even more this point let us consider a real program state¹ [18] used commonly for testing WCET tools. The program is generated automatically and its main feature is the huge amount of infeasible paths. We try to build the worst possible scenario by instrumenting the program and adding $x=0$ at the first statement of the program where x is a fresh variable, and then adding the condition `if (x>0) error()` at the end. An AR tool should add only the predicate $(x = 0)$ to prove that the program is bug free. However, an actual evaluation using BLAST shows some significant performance degradation as it may not always choose the right predicate, resulting in 21 predicates discovered in 74 seconds on Intel 2.33Ghz 3.2 GB (our algorithm takes 88 seconds). This experiment exhibits the worst possible scenario for our approach and also illustrates another potential limitation of AR. If the abstract error path has more than one infeasibility reason, then existing refinement techniques have difficulties in choosing the right refinement. Synergy-like tools mitigate this problem but introduce other challenges as discussed above.

4 Formalities

Here we briefly model a program as a transition system and formalize the proof process as one of producing a closed tree of the transition steps. It is convenient to use the formal framework of *Constraint Logic Programming (CLP)* [15], which we outline as follows.

The *universe of discourse* is a set of terms, integers, and arrays of integers. A *constraint* is written using a language of functions and relations.

An *atom* is of the form $p(\tilde{t})$ where p is a user-defined predicate symbol and the \tilde{t} a tuple of terms. A *rule* is of the form $p(k, \tilde{x}) :- p(k', \tilde{x}') \wedge \tilde{c}$ where the atom $p(k, \tilde{x})$ is the *head* of the rule, and the atom $p(k', \tilde{x}')$ and the (conjunction of) constraint \tilde{c} (possibly relating the variables \tilde{x} and \tilde{x}') constitute the *body* of the rule. Here both k and k' are positive numbers denoting program points or the special constant *error* to denote an error location. We may omit either the atom or the constraint from the body. A *goal* has exactly the same format as a body of a rule. Given a goal $G: p(k, \tilde{x}) \wedge \phi$, we denote by $cons(G)$ the constraint ϕ or *true* when ϕ is empty.

Each CLP rule represents a transition in the program¹. For example, given a program fragment with two variables x and y , the assignment `5: x = y+1 6:` is represented as the rule $p(5, x, y) :- p(6, x', y') \wedge y' = y \wedge x' = y + 1$. For a conditional `6: if (x>0) 7:`, we represent the transition between $\langle 6 \rangle$ and $\langle 7 \rangle$ by the rule $p(6, x, y) :- p(7, x', y') \wedge y' = y \wedge x' = x \wedge x > 0$.

A *substitution* simultaneously replaces each variable in a term or constraint into some expression. We specify a substitution by the notation $[\tilde{e}/\tilde{x}]$, where \tilde{x} is a sequence x_1, \dots, x_n of variables and \tilde{e} a list e_1, \dots, e_n of expressions, such that x_i is replaced by e_i for all $1 \leq i \leq n$. Given a substitution θ , we write as $e\theta$ the application of the substitution to an expression e . A *renaming* is a substitution which maps variables into variables. A *grounding* is a substitution which maps each variable into a value in its domain. A *ground instance* of a constraint, atom and rule is defined in the obvious way.

¹ For lack of space, we refer readers to [17] and its references for more details about the translation from transition systems to CLP programs.

Given a goal $G \equiv p(k, \tilde{x}) \wedge \Psi(\tilde{x})$, $\llbracket G \rrbracket$ is the set of the groundings θ of the primary variables \tilde{x} such that $\exists \Psi(\tilde{x})\theta$ holds. A goal $\overline{G} \equiv (k, \tilde{x}), \overline{\Psi}(\tilde{x})$ *subsumes* another goal $G \equiv p(k', \tilde{x}') \wedge \Psi(\tilde{x}')$ if $k = k'$ and $\llbracket \overline{G} \rrbracket \supseteq \llbracket G \rrbracket$. Equivalently, we say that \overline{G} is a *generalization* of G . We write $G_1 \equiv G_2$ if G_1 and G_2 are generalizations of each other.

We use the notion of *reduction* to represent symbolic strongest postcondition operation. Let a rule $R : p(k, \tilde{x}) :- p(k', \tilde{x}') \wedge \tilde{c}$ belong to a CLP program. Given a goal $G : p(k, \tilde{x}_i) \wedge \Psi$ with variables disjoint from R , a *reduct* or *derivation* of G using R (denoted $\text{reduct}_R(G)$) is the goal $p(k', \tilde{x}_{i+1}), \Psi \wedge \tilde{c}[\tilde{x}_i/\tilde{x}][\tilde{x}_{i+1}/\tilde{x}']$. A derivation sequence (path) is a sequence of goals G_0, G_1, \dots where $G_i, i > 0$ is a reduct of G_{i-1} .

A goal $G : p(k, \tilde{x}) \wedge \tilde{c}$ is called *terminal* if there are no applicable rules to perform reduction on it, and it is called *looping* if it is derived from another goal with the same k (called its *looping parent*) through one or more reduction steps. A goal is *infeasible* if its constraints are unsatisfiable, and a derivation sequence is so called when it ends in an infeasible goal.

5 Algorithm: Minimax

As mentioned above, there is an obvious strategy for dealing with loops by using iterative deepening on the level of loop unrolling, and in each iteration, to generate loop invariants. In this section, we present an algorithm that performs unrolling in an *intelligent* manner, using information about why a particular path does not suffice. In this regard, there is similarity to CEGAR where, if a candidate loop invariant is found insufficient (too weak), the refinement process takes into account the *reason* for this insufficiency in order to arrive at the next refinement.

Our algorithm maintains knowledge about a state (goal) $G \equiv p(k, \tilde{x}) \wedge c_1 \wedge \dots \wedge c_n$ by means of a vector $v \equiv \langle \alpha^1, \dots, \alpha^n \rangle$ where each α^i is an *annotation* of one of the following kinds:

- a *max* annotation, indicating that the constraint c_i *must be kept*
- a *min* annotation, indicating that the constraint c_i *must be deleted*, or
- a *neutral* annotation.

Denote the i -th annotation in v by α_v^i . Let c be a constraint, its annotation is denoted $\alpha_v(c)$. $\text{neut}(\tilde{c})$ denotes a vector $\langle \text{neutral}, \dots, \text{neutral} \rangle$ of the same length as \tilde{c} . We write $\text{conflict}(v_1, v_2)$ if $\exists 1 \leq i \leq \min\{\text{length}(v_1), \text{length}(v_2)\}$ such that $(\alpha_{v_1}^i = \text{min})$ and $(\alpha_{v_2}^i = \text{max})$.

A pair (G, v) where the state $G \equiv p(k, \tilde{x}) \wedge \tilde{c}$ is called an *annotated state*. The meaning of an annotated state $\sigma = (G, v)$ is obtained in two ways. A *max* interpretation σ_{max} is the state obtained by deleting all but the *max*-annotated constraints in \tilde{c} . Dually, a *min* interpretation σ_{min} is the state obtained by including all but the *min*-annotated constraints in \tilde{c} . For example, given an annotated state $\sigma : p(5, x_1, x_2, x_3) \wedge x_1 = 1 \wedge x_2 = 2 \wedge x_3 = 3, \langle \text{min}, \text{neutral}, \text{max} \rangle$, σ_{max} and σ_{min} are, respectively the two states $p(5, x_1, x_2, x_3) \wedge x_2 = 2 \wedge x_3 = 3$ and $p(5, x_1, x_2, x_3) \wedge x_3 = 3$. Note $\text{cons}(\sigma_{\text{min}})$ is weaker than $\text{cons}(\sigma_{\text{max}})$.

The use of vectors is an efficient way for computing interpolants. *max*-annotated constraints of an annotated state $\sigma = (G, v)$ must be kept to preserve some infeasible paths in the derivation tree emanating from σ . Given an infeasible annotated state σ'

derived from σ , we minimally *max*-annotate the constraints in σ' (some of which are constraints of σ since they share the vector) such that the infeasibility is maintained. In this way we immediately obtain an abstraction at σ (that is, σ_{max}) by the *max* annotations generated at σ' without performing *weakest precondition* propagation or some approximation of it. σ_{max} subsumes G yet it entails the infeasibility of σ' and therefore is an interpolant. The final abstraction at σ is a conjunction of the interpolants returned by the children, and this is easily obtained by the conjunction of all *max*-annotated constraints at σ after the subtree is traversed.

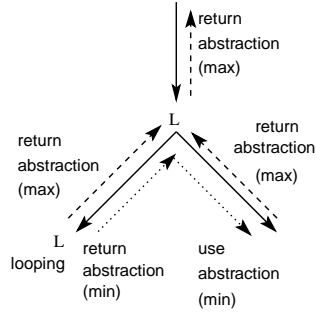


Fig. 4. Min-Max

The algorithm operates on annotated states. Its depth-first traversal is outlined in Fig. 4. When encountering a loop (point L in Fig. 4) a loop invariant is produced by weakening the constraints at L by minimally *min*-annotating its state. This weakening is then applied in the forward execution of the points beyond the L . This abstraction, however, is not the final abstraction that is used to subsume other states since it still can be weakened further as some constraints may not contribute to the infeasibility or subsumption of descendant states. The final abstract state (in L or elsewhere) is computed by propagating *max* annotations backward in post-order manner. *Max* annotations are produced by interpolation at points where infeasibility and subsumption are found (Lines 2, 5 and 10 in Fig. 5).

Before detailing the algorithm of Fig. 5 we first explain its main components.

Interpolation. If σ is $(p(k, \tilde{x}) \wedge \tilde{c}, v)$ and $\text{cons}(\sigma_{min}) \wedge \phi$ is unsatisfiable, $\text{interpolate}(\sigma, \phi)$ returns an annotation v' which has the same length as v (and \tilde{c}), satisfying the following:

1. $\forall c \in \tilde{c} : \alpha_v(c) \in \{\min, \max\} \Rightarrow \alpha_{v'}(c) = \alpha_v(c)$,
2. $\forall c \in \tilde{c} : \alpha_v(c) = \text{neutral} \Rightarrow \alpha_{v'}(c) \in \{\text{neutral}, \max\}$, and
3. $\sigma' = (p(k, \tilde{x}) \wedge \tilde{c}, v') \Rightarrow (\text{cons}(\sigma'_{max}) \wedge \phi \text{ unsatisfiable})$

v' is computed by adding the fewest *max* annotations to neutral annotations in v , thus representing a computation of an *interpolant*: σ'_{max} maintains the unsatisfiability (consequence) of σ_{min} yet it has less constraints (more general). For example, consider the annotated state $\sigma : (p(k, x_1, x_2) \wedge \tilde{c}, \langle \min, \max, \text{neutral}, \text{neutral} \rangle)$ with \tilde{c} be $x_1 > 3 \wedge x_1 = y_1 + 1 \wedge y_1 = 2 \wedge x_2 = 0$, and a constraint $\phi : x_1 < 0$. where Here, σ_{min} is unsatisfiable. Then $\text{interpolate}(\sigma, \phi)$ produces the vector $v' : \langle \min, \max, \max, \text{neutral} \rangle$. That is, the third constraint's annotation is changed from *neutral* to *max* such that $\sigma'_{max} \wedge \phi$ maintains the unsatisfiability.

Subsumption and Loop Invariants. An essential feature of our algorithm, existing also in AR methods, is the ability of blocking the forward search traversal of an annotated state σ if there exists another state σ' already processed such that the state of σ entails the state associated with σ' . During the symbolic traversal there are two kinds of subsumptions.

Parent-Child: assume that σ' is a looping ancestor of σ . Here σ' would be of the form $(p(k, \tilde{x}') \wedge \tilde{c}_1, v_1)$ and σ of the form $(p(k, \tilde{x}) \wedge \tilde{c}_1 \wedge \tilde{c}_2, v)$, where $v = v_1 \cdot v_2$ with v_2 of the same length as \tilde{c}_2 . Since we would like to unroll as few times as possible, the algorithm forces (if possible) parent-child subsumption by computing the strongest path-based loop invariant. Therefore, v_1 can be replaced with a vector \overline{v}_1 of the same length where some *neutral* annotations (those that are not individually invariant) in v_1 are transformed to *min* annotations in \overline{v}_1 such that σ'_{min} subsumes σ_{min} . The function $\text{invariant}(\sigma, \sigma')$ returns the vector $\overline{v}_1 \cdot v_2$ if subsumption holds. Otherwise, the parent-child subsumption is not possible and the algorithm returns \perp . This our mechanism to lazily unroll loops.

Sibling-Sibling: assume now the state σ' has been already processed and stored in a *memo table*, \mathcal{M}_T . The condition here is that the current state associated to σ entails the interpolant associate to σ' . That is, σ'_{max} subsumes σ_{min} . This test is done by the function $\text{subsumed}(\mathcal{M}_T, \sigma)$ in the algorithm. If the test holds, this function also returns a *subsuming state* σ^{sub} . Otherwise, \perp .

For σ^{sub} we need to distinguish two subcases. If σ' is out of the scope of a loop, then $\sigma^{sub} = \sigma'$. Otherwise, as in the case of parent-child subsumption, we may need to convert some *neutral* annotations into *min* annotations to communicate ancestors the conditions under the subsumption took place. In particular, those neutral annotations which if had been *max* annotations then subsumption would not have held.

Merging Vectors. We use two operations for merging both *min* and *max* annotations. Given two vectors v_1 and v_2 :

$\text{mergemin}(v_1, v_2)$: if the condition $\forall 1 \leq i \leq \min\{\text{length}(v_1), \text{length}(v_2)\} : \alpha_{v_1}^i = \min \Rightarrow \alpha_{v_2}^i \in \{\text{neutral}, \min\}$ holds then it returns a vector v satisfying $\forall 1 \leq i \leq \text{length}(v_1) : (\alpha_{v_1}^i = \min \Rightarrow \alpha_v^i = \min) \wedge (\alpha_{v_1}^i \neq \min \Rightarrow \alpha_v^i = \alpha_{v_2}^i)$. Otherwise, the function returns \perp .

$\text{mergemax}(v_1, v_2)$: returns always a vector v satisfying $\forall 1 \leq i \leq \max\{\text{length}(v_1), \text{length}(v_2)\} : ((\alpha_{v_1}^i = \max \vee i > \text{length}(v_2)) \Rightarrow \alpha_v^i = \alpha_{v_1}^i) \wedge ((\alpha_{v_1}^i \neq \max \vee i > \text{length}(v_1)) \Rightarrow \alpha_v^i = \alpha_{v_2}^i)$.

The Minimax routine takes as inputs the depth \mathcal{D} of the symbolic tree, a current annotated state σ , and the table C_T to record the ancestor states that can potentially become the looping parent of the current state. There is a global table, \mathcal{M}_T , to store the interpolants already computed. The execution starts with some $\mathcal{D} = 0$, σ_{init} which is neutral, and an empty C_T . The memo table, \mathcal{M}_T , is also initially empty.

Line 2 handles the case when the state is infeasible. Here, *max* annotations are created using the procedure *interpolate* to indicate constraints that are needed in order to preserve unsatisfiability of the constraints.

Lines 4–7 handle the case when *error* program point is visited with feasible σ_{min} . In case σ is itself feasible (\tilde{c} is satisfiable), we have found a real error, and the algorithm aborts (Line 4). In case σ is infeasible, we have found a spurious state, which is visited due to the weakening caused by *min* annotations. At Line 5 we compute *max* annotations such that the infeasibility is preserved. At Line 6 we compute the shallowest depth value such that the conflict occurs, from which we are to restart. We then add the computed *max* annotations to the input vector and returns the resulting vector together with a CONFLICT status and the computed depth (Line 7).

```

Minimax( $\mathcal{D}, \sigma, C_T$ ) returns (OK, $a,b$ ) or (CONFLICT, $a,b$ ) with vector  $a$  and integer  $b$ 
  let  $\sigma$  be  $(\mathcal{G}, v)$  and  $\mathcal{G}$  be  $p(k, \tilde{x}) \wedge \tilde{c}$ 
  switch( $\sigma$ )
1:   case  $\text{cons}(\sigma_{\min})$  unsatisfiable:
2:     return (OK, interpolate( $\sigma, \text{true}$ ), 0)
3:   case  $k = \text{error}$ :
4:     if ( $\tilde{c}$  is satisfiable) abort
5:      $v' := \text{interpolate}((\mathcal{G}, \text{neut}(\tilde{c})), \text{true})$ 
6:      $d := \min\{l \mid (l, (\mathcal{G}', v'')) \in C_T \text{ and } \text{conflict}(v'', v')\}$ 
7:     return (CONFLICT, mergemax( $v', v$ ),  $d$ )
8:   case  $\mathcal{G}$  is terminal: return (OK,  $v$ , 0)
9:   case There is  $\sigma^{\text{sub}} = \text{subsumed}(\mathcal{M}_T, \sigma)$  and  $\sigma^{\text{sub}} \neq \perp$  :
10:    return (OK, interpolate( $\sigma, \neg \text{cons}(\sigma_{\max}^{\text{sub}})$ ), 0)
11:   case  $S = \{\sigma' \mid (l, \sigma') \in C_T \text{ and } \sigma' \text{ looping parent of } \sigma\} \neq \emptyset$ :
12:    foreach  $\sigma' \in S$ 
13:      if ( $v' = \text{invariant}(\sigma, \sigma')$  and  $v' \neq \perp$ ) return (OK,  $v'$ , 0)
14:    goto default
  default :
15:     $v' := v, \sigma := (\mathcal{G}, v')$ 
16:    foreach  $\mathcal{G}'$  in  $\text{red}_{\min}^+(\sigma) \dots \text{red}_{\min}^-(\sigma)$ 
17:      let  $\text{cons}(\mathcal{G}')$  be  $\tilde{c} \wedge \tilde{c}'$ 
18:       $(\text{Status}, v'', d) := \text{Minimax}(\mathcal{D} + 1, (\mathcal{G}', v' \cdot \text{neut}(\tilde{c}')), C_T \cup \{(\mathcal{D}, \sigma)\})$ 
19:       $v''' = \overline{\text{wp}}((\mathcal{G}', v''), \tilde{c}')$ 
20:      if ( $\text{Status} = \text{CONFLICT}$ )
21:        if ( $d = \mathcal{D}$ )
22:           $\mathcal{M}_T := \mathcal{M}_T \setminus \{\sigma' \mid \sigma' = (\mathcal{G}'', \cdot) \text{ and } \mathcal{G}'' \text{ derived from } \mathcal{G}\}$ 
23:          return Minimax( $\mathcal{D}, (\mathcal{G}, v'''), C_T$ )
24:          else return (CONFLICT,  $v''', d$ )
25:         $v' := \text{mergemax}(v''', \text{mergemin}(v''', v'))$ 
26:       $\mathcal{M}_T := \mathcal{M}_T \cup \{(\mathcal{G}, v')\}$ 
27:    return (OK,  $v'$ , 0)

```

Fig. 5. The Minimax Algorithm

Line 8 is selected if the end of the path is reached. Here it is not necessary to add either *min* or *max* annotations as looping points or infeasible/subsumed states can no longer be reached, and we therefore return OK and the input annotation itself.

Lines 9–10 handle the case if the current state is subsumed by another state already memoed in \mathcal{M}_T . Recall the notion of subsuming state σ^{sub} explained previously. Here we return OK with a vector with more *max* annotations than the input vector v needed to ensure the entailment (unsatisfiability of the negation of the constraints) of the abstraction of the subsuming state ($\sigma_{\max}^{\text{sub}}$).

Lines 11–14 handle the case when the current state is looping. Here we attempt to compute a path-based loop invariant using *min* annotations that are produced by calling invariant subprocedure (Line 13) in order to force parent-child subsumption. If invariant fails to produce the abstraction, we continue to the default case (Line 14).

The default case at Lines 15–27 performs one symbolic execution step. We first formalize functions used in Line 16. Let σ be (\mathcal{G}, v) , we denote by $\text{red}_{\min}^+(\sigma)$ the set $\{\mathcal{G}' \mid \exists R : \text{cons}(\text{reduct}_R(\sigma_{\min})) \text{ is satisfiable} \wedge \mathcal{G}' = \text{reduct}_R(\mathcal{G})\}$. Similarly, we denote

by $red_{min}^-(\sigma)$ the set $\{G' | \exists R : cons(reduct_R(\sigma_{min})) \text{ is unsatisfiable} \wedge G' = reduct_R(G)\}$. In essence, $red_{min}^+(\sigma)$ ($red_{min}^-(\sigma)$) is the set of reducts of G such that, using the same rules, the reduction of σ_{min} is feasible (infeasible). In this way, the loop in Line 16 makes us prioritize transitions that are feasible, possibly due to *min* abstraction. This is important to not generate *max* annotations that restricts abstraction too early resulting in failure to discover loop invariants later (inability to convert *max* to *min* annotations at Line 13). Then, the loop iterates in sequence over the reducts, performing recursive calls to Minimax (Line 18). The result, for each one, is a triple $(Status, v'', d)$. v'' here contains *max* annotations that specify how the current state need to be abstracted. At Line 19 we compute the abstraction for the current state based on the annotation returned using the function \overline{wp} , which denotes an approximation of the weakest precondition. In our framework, this can be trivially done, without calling the theorem prover, by cutting off the last $|\tilde{c}'|$ elements of v'' . (Here \tilde{c}' are the constraints added by the reduction.)

If *Status* is CONFLICT with depth d (produced at Line 7), we know that somewhere during the recursive call, a conflict occurred. If the depth d is equal to the current depth \mathcal{D} , then this is the topmost point where the conflict is originated. In addition, we know that v''' (the vector after calling \overline{wp}) is the same as v , but with some *max* annotations replacing *min* annotations. In essence, we “lock” such annotations. More importantly, this may result in failure to create loop invariant at Line 13 later. At Line 22 we need to clean the memo table for those states derived from the current input state. We then perform another recursive call (Line 23) as a replacement for the current call (without making any transition step), and using v''' to propagate the locked annotations. If the current depth is not equal to the conflict depth, we simply propagate the conflict to the parent (Line 24).

Finally, Line 25 combines the vectors returned by each descendant, and after all vectors are merged, Line 26 stores in the memo table the interpolant for the current goal.

We conclude this section by mentioning that the central step of deleting constraints, the effect of a *min* annotation, can in fact be relaxed to some other mechanism that abstracts the state at hand. Instead of deleting a constraint, one could *transform* a constraint. For example, one could apply a process of “slackening” to equations $x = y$ to obtain an inequality, either $x \leq y$ or $x \geq y$. This kind of abstraction is in fact employed in the BLAST system which we benchmark against, but at this time, we do not use for our own experimental results. Even more generally, we could replace not one but a collection of constraints by another collection which is entailed by the original collection.

6 Experimental Evaluation

We implemented our prototype TRACER modelling the C heap using the theory of arrays with alias analysis to partition and inlining functions. We ran TRACER on several programs instrumented with safety properties and compare with BLAST [6]². We downloaded all programs from [5] already instrumented with safety conditions, and together with a script which runs those programs with the most favorable system options.

² We tried with ARMC but we were only successful to run on tcas and statemate but timeout expired in both cases after 30m and 1h, respectively.

The results are summarized in Table 6. We present two sets of numbers: for BLAST the number of discovered predicates (P) the total time in seconds (T), and for TRACER, our prototype tool, the number of nodes of the exploration tree (S) and also the total time in seconds (T). Although the number of discovered predicates and nodes of the exploration tree are not comparable they are shown to provide an idea about the hardness of the proof.

Program	LOC	BLAST (AR)		TRACER (AL)	
		P	T	S	T
qpmouse	400	4	0.42	974	0.42
tlan	8069	14	17.10	4382	5.78
cdaudio	8921	*	*	6258	10.53
diskperf	6984	92	82.3	3326	8.21
floppy	8570	*	*	3124	6.47
kbfiltr	5931	45	44.03	1392	2
serial	10380	*	*	59597	328.6
tcas-1a-safe	394	23	3.6	6029	6.97
tcas-1b-safe		56	78.35	6050	6.77
tcas-2a-safe		22	3.25	6029	6.74
tcas-3b-safe		39	15.68	6017	6.63
tcas-5a-safe		31	10.29	6029	6.36
tcas-2b-unsafe		40	17.46	91	0.01
tcas-3a-unsafe		25	18.96	243	0.16
tcas-4a-unsafe		45	14.44	243	0.15
tcas-4b-unsafe		36	6.44	91	0.01
tcas-5b-unsafe		54	40.31	91	0.02

Fig. 6. BLAST Benchmarks on Intel 2.33Ghz 3.2GB

and 156 predicates, respectively on Pentium 2.4Ghz 512Mb.

Special mention deserves the cases where the programs were proved unsafe. In these cases, TRACER found a real counterexample much faster than BLAST. The reason is that TRACER blocks infeasible paths and then finds very quick the real error. BLAST will spent some time performing refinements and traversing space which are irrelevant to the real error path. Nevertheless, this is an example where we believe that Synergy-like tools using test cases would perform as ours since DART could also find the real error path faster.

7 Concluding Remarks

We extended Abstraction Learning, an interpolation-based symbolic execution method, to automatically handle unbounded loops. The algorithm is an intelligent unrolling process by classifying into *min* and *max* constraints. The *min* constraints are those which must be abstracted in order to achieve subsumption and loop invariance, while the *max* constraints are those which must not be abstracted so as to detect infeasible paths and also to preserve safety. The idea is to have as few of these two kinds of constraints as

In summary, TRACER is competitive with BLAST in most of the benchmark examples, sometimes much faster. However, there are two programs where BLAST is faster (tcas-1a, and tcas-2a). We believe the main reason is that TRACER does perform some extra work due to unnecessary infeasible paths. Nevertheless, the numbers show that the differences are not significant.

Note that programs such as cdaudio, floppy, and serial are annotated with the symbol '*' in the BLAST column which means that BLAST raised an exception and aborted. Therefore, we were not able to verify those programs using BLAST. Although we could not contact BLAST authors we are aware that cdaudio and floppy have been proved safe in [13] after 21m59s and 11m17s discovering 196

possible. We discussed the relative merits of ours and AR-based methods using academic examples. We also evaluated our prototype, TRACER, against BLAST, the most advanced system available to us, using real programs. The results show competitive performance, with some examples showing significant improvement. In all cases, the results show that eagerly detecting infeasible paths can be efficient.

References

1. T. Ball, B. Cook, V. Levin, and S. K. Rajamani. SLAM. In *IFM'2004*.
2. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI'01*, pages 203–213.
3. N. E. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons. Proofs from tests. In *ISSTA '08*, pages 3–14.
4. D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Path Invariants. In *PLDI'07*, pages 300–309.
5. D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. BLAST. URL <http://mtc.epfl.ch/software-tools/blast/index-epfl.php>.
6. D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The Software Model Checker BLAST. *Int. J. STTT*, 9:505–525, 2007.
7. S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *IEEE Transactions on Software Engineering*, 30(6):388–402, June 2004.
8. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. CounterExample-Guided Abstraction Refinement. In *CAV'00*.
9. P. Cousot, P. Ganty, and J.F. Raskin. Fixpoint-guided abstraction refinements. In *SAS'07*, pages 333–348.
10. P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *PLDI*, pages 213–223, 2005.
11. P. Godefroid, A. V. Nori, S. K. Rajamani, and Sai Deep Tetali. Compositional may-must program analysis: unleashing the power of alternation. *POPL'10*, pages 43–56.
12. B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. Synergy: a new algorithm for property checking. In *SIGSOFT '06/FSE-14*, pages 117–127, 2006.
13. T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *31st POPL*, pages 232–244. ACM Press, 2004.
14. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *29th POPL*, pages 58–70. ACM Press, 2002. SIGPLAN Notices 37(1).
15. J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *J. LP*, 19/20:503–581, May/July 1994.
16. J. Jaffar, A. E. Santosa, and R. Voicu. Efficient memoization for dynamic programming with ad-hoc constraints. In *23rd AAAI*, pages 297–303. AAAI Press, 2008.
17. J. Jaffar, A. E. Santosa, and R. Voicu. An interpolation method for CLP traversal. In *15th CP*, volume 5732 of *LNCS*. Springer, 2009.
18. Mälardalen WCET research group benchmarks. URL <http://www.mrtc.mdh.se/projects/wcet/benchmarks.html>, 2006.
19. K. L. McMillan. Lazy annotation for program testing and verification. In *CAV'10*.
20. A. Podelski and A. Rybalchenko. ARMC. In *PADL'07*.
21. Hassen S. Model checking guided abstraction and analysis. In *SAS '00*.